SEVENTH FRAMEWORK PROGRAMME
Collaborative Project
Research Infrastructures

TEXT - Towards EXaflop applicaTions

Contract no: RI261580
Project Officer: Mr. Leonardo Flores

Deliverable Number: D6.3
Deliverable Title: Application Porting from External developers
Delivery Date: 31/08/2012

Dissemination Level: PU
Lead Editor: Jesus Labarta, BSC

Beneficiaries involved: BSC, USTUTT, JUELICH, UEDIN, FORTH, UNIMAN, UPPA, UJI and IBM

Contributors:

Jesus Labarta (SBC)
Iain Bethune (EPCC)
Marta Garcia (BSC)
Victor Lopez (BSC)
Grahan Rilley (UniMAN)

Internal Reviewer(s):

Eduard Ayguade (BSC)
Rosa M. Badia (BSC)

Version History:

Version	Date	Changes	Author
0.1	01/08/2012	First Structure Draft	Jesus Labarta
0.2	16/08/2012	Initial contributions (Hydro, CP2K)	Jesus Labarta, Iain Bethune
0.3	26/08/2812	Initial contribution (GROMACS, GADGET, MRGENESIS, ShallowWaters)	Marta Garcia, Victor Lopez, Graham Riley
0.4	03/09/2012	First draft for internal comments	Jesus Labarta
1.0	07/09/2012	Final version	Jesus Labarta

Deliverable Summary:

Deliverable Number:	D6.3
Deliverable Title:	Application Porting From External Developers
Lead beneficiary number:	5
Estimated indicative person months:	18
Nature:	R
Dissemination level:	PU
Delivery date:	August 31, 2012

Contents

1 Executive Summary

Different developers outside the project or applications not initially considered in it have been identified and addressed in order to assess how the MPI/SMPSs model could be used in those cases. Different levels of depth have been actually been carried out, and in some of the cases only a part of the application was targeted. The study mostly focused on widening the identification of methodological issues and feedback on the suitability of the model and the functionality of its implementations.

This deliverable reports on the experience with 6 apps/codes: Hydro, GROMACS, GADGET, CP2K, MRGENESIS and ShallowWaters.

Very interesting feedback has been obtained addressing both methodological programming practices and the characteristics of the SMPSs/OmpSs model and implementation.

1 Objectives

This deliverable reports on the results of the activities carried out to apply or assess the applicability of the MPI/SMPSs model to other codes not part of the applications initially contributed by project partners.

2 List of external users codes

The following table enumerates the list of external applications

Application	**Code developer**
Hydro	Romain Teyssier, Guillaume Colin de Verdiere (CEA)
GROMACS	Erik Lindahl (KTH)
GADGET	(Heidelberg)
CP2K	Iain Bethune (EPCC)
MRGENESIS	M.A. Aloy (UPV)
Shallow Water	Graham Riley. (U. Manchester)

The following sections describe each application, its structure and parallelization approach, results and feedback on MPI/SMPSs. For each of them we very briefly describe the application, the parallelization approach, some results and feedback from the experience.

3 Hydro

3.1 Short application description

Hydro[7] is a proxy benchmark of the RAMSES[8] application that solves a large-scale structure and galaxy formation. Hydro uses a rectangular 2D space domain split in blocks. Hydro solves compressible Euler equations of hydrodynamics, is based on finite volume numerical method using a second order Godunov scheme for Euler Equations and a Riemann solver computes numerical flux at the interface of two neighboring computational cells.

Different version of Hydro have been produced by CEA for different programming models and target platforms (C, Fortran, MPI with 1D and 2D decompositions, OpenMP, CUDA, HMPP,…). We looked at the C MPI versions to introduce OmpSs within it.

3.2 Parallelization approach

We followed two approaches. In the first one we took as reference a fine grain OpenMP version, while the second one started from a newer C version that had been produced by CEA as an intermediate step towards the manual introduction or CUDA.

The general source code structure for the application is as follows

```
for (iter = 0; iter < NITERS; iter++) {

  for (dimension = 0; dimension < 2; dimension++) {

     make_boundary();

     Allocate local variables

     for (j = Hmin; j < Hmax; j++) {
       gatherConservativeVars(dimension,uold,localvars,…);
       constoprim(dimension, localvars,… );
       equation_of_state(dimension, localvars,…);
       slope (dimension, localvars,…);
       trace(dimension, localvars,…);
       qleftright(dimension, localvars,…);
       riemann(dimension, localvars,…);
       cmpflx(dimension, localvars,…);
       updateConservativeVars(dimension,localvars, uold,…);
     }

    Free local variables
`
```

Figure 1: skeleton structure of Hydro

For each time step iteration, two traversals of the matrix are performed along the row and column dimensions respectively. Each of the internal subroutines has a loop that updates either row wise or column wise (depending on the value of dimension) the main data structures.

The make_boundary routine is where MPI communication takes place. The application scales very well at the MPI level for the problem sizes and core counts that we were looking at, so our main focus has been on the OmpSs parallelization of the computational part between communications.

3.2.1 Bottom-up approach

Our first effort followed a bottom up approach. The iterations of each of the inner loops that traverse the matrix either row wise or column wise are independent so the loops can be parallelized with either OpenMP or OmpSs. One feature of OpenMP that is not supported by the task based OmpSs is the possibility to specify that a loop is parallel and leave to the runtime the splitting into tasks. This means that in OmpSs we had to strip-mine a loop manually and associate a task to the inner loop, which is somewhat cumbersome.

This fine grain approach does have two issues. First, the granularity of the tasks can be very small, thus overheads become significant and performance is not good. Second it tends to derive in a pure fork join approach where each loop is considered in isolation, without exploiting its potential concurrency with other loops.

In order to reduce some overhead in OpenMP, the approach implemented in the available fine grain OpenMP version was to move the parallel pragma to the main routine and relying a lot on the single pragma and orphaning for pragmas inside the routine compute routine (in the sketch of Figure 1 only the j loop requires a for pragma). More important than that, another practice used in this code was to declare as thread private important data structures that contain pointers that are then allocated and freed by each thread further expanding the amount of thread private data. It all makes it difficult to identify in the source code to which data are we really referring to.

This approach parallelizes along only one dimension (alternatively rows and columns). In order to minimize the surface to volume ratio, and minimize the amount of data movement (invalidations and cache misses) caused by alternating rows and columns a version of a 2D parallelization approach was also available. In it, a synchronization array and the OpenMP flush pragma was used to ensure proper order of the computations. This code behaves better but is certainly more complex to understand and maintain.

We do believe that these practices are allowed (actually encouraged?) by the OpenMP model and methodology and end up generating codes difficult to understand and maintain.

The approach to increase granularity in the OmpSs version was to strip-mine the internal loops inside the routines called in Figure 1 and perform loop interchange across the subroutine boundary resulting in a code sketched in Figure 2. This keeps the one dimensional parallelization approach where each task applies to a set of rows (or columns) the specific function it represents. Unfortunately, the fact that the dependency chain within the j iteration happens to be very linear results in no real benefit from instantiating many tasks if they end up executing serially. Furthermore, at the end of the j loop, before the change in dimension, an OmpSs barrier (taskwait pragma) has to be inserted as the accesses become strided and the dependences were not properly computed by the version of OmpSs used. Even if we have increase the granularity, the fact that a single level of parallelism is used implies that all task instantiations are serialized thus requiring large block sizes to reduce the number of tasks and amortize the instantiation cost.

```
for (iter = 0; iter < NITERS; iter++) {

  for (dimension = 0; dimension < 2; dimension++) {

     make_boundary();

     Allocate local variables

     for (j = Hmin; j < Hmax; j += Hstep) {
       for (ii = iinf; ii < isup; ii += iBS) {
           iimax = ii+iBS > isup? isup : ii+iBS;
           #pragma omp task input(*uold) output(u[ii])
           gatherConservativeVars(slice, uold, u, …);
       }

       for (ii = iinf; ii < isup; ii += iBS) {
           iimax = ii+iBS > isup ? isup : ii+iBS;
           #pragma omp task input(u[ii]) inout(q[ii]) output(e[ii])
           constoprim(slice, u, q, e, …);
       }

       for (ii = iinf; ii < isup; ii += iBS) {
           iimax = ii+iBS > isup ? isup : ii+iBS;
           #pragma omp task inout(qID[ii]) output(c[ii])
           equation_of_state(slice, qID, e, qIP, c, ii, iimax, …);
       }
       ……..
   }

}
```

Figure 2: Bottom-up OmpSs parallelization

A problem we found was that the code was relying a lot on indexing arithmetic. This is probably a far too frequent practice that makes the codes difficult to understand for a new developer. This was certainly a problem to determine the access patterns required for the pragmas. Also very important is the fact that it also makes impossible to properly specify in the OmpSs syntax (even with region support) the precise part of an array accessed by a task. Certainly we do consider that a bit more of discipline in programmers precisely specifying the types of variables would be useful, not only for OmpSs but for programmability and maintainability purposes in general.

3.2.2 Top down approach

Looking at the coarse grain structure of the algorithm we see that iterations of the j loop are independent so they can be split in blocks and tasks used to compute each such block.

In our case, a code with the structure showed in Figure 3 results. The very linear sequence of dependences between the different routines within each iteration makes not sensible to split the loop body in different tasks from the point of view of performance. As we can see, only one loop is parallelized, thus resulting in a 1D parallelization and given that the main matrix (uold) is traversed in different directions for the alternate values of the dimension loop, complex dependences arise between two such iterations. A barrier (taskwait) is thus used to enforce such dependences.

We have manually privatized several large data structures dynamically allocated whose use is local to each block of iterations.

This structure is essentially the same as the one in the original OpenMP code, although we believe the privatization approach is cleaner in the task based version. A general top down approach for other codes might result in parallelism being extracted at different levels where

different functions or loops could be executed concurrently exploiting the OmpSs dependence driven execution.

```
for (iter = 0; iter < NITERS; iter++) {
  for (dimension = 0; dimension < 2; dimension++) {
     make_boundary();
     for (j = Hmin; j < Hmax; j += Hstep) {
        #pragma omp task concurrent (*uold)
        {
           Declare & Allocate local variables
           (u, qleft, qright, qgdnv, flux, sgnm, q, dq, qxm, qxp, e, c)
           gatherConservativeVars(slice, uold, u, …);
           constoprim(slice, u, localvars,…);
           equation_of_state(slice, localvars,…);
           slope (slice, localvars.…);
           trace(slice, localvars, …);
           qleftright(slice, localvars, qleft, qright, …);
           riemann (qleft, qright, qgdnv, sgnm,…)
           cmpflx(slice, qgdnv, flux, …);
           updateConservativeVars(slice, uold, u, flux,…);
           Free local variables
        }
     }
     #pragma omp taskwait
  }
```

Figure 3: Top down coarse grain taskification

3.2.3 Task splitting

The next objective was to start introducing CUDA code, by leveraging the CUDA kernels from the original MPI+CUDA version. Taking as reference the coarse grain parallelization obtained in the previous section and the available CUDA kernels the first observation is that these correspond to only a part of the whole computational chain within the main loop body.

Thus, a preliminary step was required. The single task of previous section had to be split into several tasks such that each of them encapsulates only the computation for which the kernel is available.

Identifying the riemman call as an expensive part of the computation for which a kernel is available, we derived a new version of the OmpSs version still targeting SMP but splitting the original tasks into three as shown in Figure 4. The nested task support in OmpSs was useful in this case. By using nesting, different first level tasks can concurrently generate second level tasks.

Beyond the actual splitting of the computation task, we also reorganized the memory management structure of the code. We identified which variables were local to each task and moved inwards their declaration and allocation. This probably goes against the dominant sequential programming practice aiming at minimizing overheads by moving outwards memory allocation and deallocation as much as possible. We do believe that such practice is very harmful as it leads to less clear and focalized code, it introduces many parallelization inhibitors (antidependences) and binds concepts (data objects) to an address space too early. Being overhead the major reason for that, we do believe that the right approach it to minimize the overhead of the memory allocation functionality. For that purpose we

developed a simple but specialized memory allocator that takes into account the repetitive allocation and freeing structure of the program heavily reusing previously allocated memory slots. We do believe that more elaborated allocators can be integrated in the runtime, this being a direction for further experimentation. Also in order to reduce the overhead, a single memory region is allocated at the entry of each task, then computing the actual pointers to the different variables needed. This functionality was made by hand in our case but could also be easily performed by the compiler as part of the private clause.

```
for (iter = 0; iter < NITERS; iter++) {
  for (dimension = 0; dimension < 2; dimension++) {
     make_boundary();
     for (j = Hmin; j < Hmax; j += Hstep) {
        #pragma omp task concurrent (*uold)
        {
           Declare & Allocate local variables (u, qleft, qright, qgdnv, flux)
           #pragma omp task output(*qleft,*qright)
           {
              Declare & allocate local variables (q, dq, qxm, qxp, e, c)
              gatherConservativeVars(slice, uold, u, …);
              constoprim(slice, u, localvars,…);
              equation_of_state(slice, localvars,…);
              slope (slice, localvars.…);
              trace(slice, localvars, …);
              qleftright(slice, localvars, qleft, qright, …);
              Free local variables
           }
           #pragma omp task input (*qleft, *qright) output (*qgdnv)
           {
              Declare & allocate local variables (sgnm)
              riemann (qleft, qright, qgdnv, sgnm,…)
              Free local variables
           }
           #pragma omp task input (*qgdnv)
           {
              Declare & allocate local variables (flux)
              cmpflx(slice, qgdnv, flux, …);
              updateConservativeVars(slice, uold, u, flux,…);
              Free local variables
           }
           #pragma omp taskwait
           Free local variables
        }
        #pragma omp taskwait
        Free local variables
  }
  #pragma omp taskwait
}
```

Figure 4: Splitting tasks to match available CUDA kernel structure

3.2.4 MPI + OmpSs + CUDA

In order to now introduce the CUDA code we substituted the invocation of Riemann in Figure 4 by a call to the function cuRiemann used in the original version to launch the kernels. We modified that function by introducing target device cuda pragmas to encapsulate the kernel invocations. Figure 5 shows the skeleton of the resulting routine.

Our objective was not to modify at all the original kernels, but we ended up doing some modifications. The original code used a programming practice, possibly promoted by CUDA itself, to declare a data structure in constant memory to which the host copies all the arguments of a kernel in a single explicit memory copy. Such a copy was done inside the original *cuRiemann* function. The result is that this function had to first pack its arguments into a local structure and copy them to the device before invoking the kernel. A second effect is that all the accesses inside the kernel were referencing fields of such global structure in the GPU memory. More important than the cumbersome need to pack the arguments and introduce additional indirections is the fact that such structure is bound to an absolute address in the GPU because of its –constant—declaration and being known to the kernel and its launcher. This practice is harmful, inhibiting for example the simultaneous execution of multiple instances of the kernel concurrently.

```
void cuRiemann(long narray, double Hsmallr, double Hsmallc, double Hgamma,
          long Hniter_riemann, long Hnvar, long Hnxyt,
          int slices, int Hnxystep,
          double *qleft,    // [Hnvar][Hnxystep][Hnxyt]
          double *qright,   // [Hnvar][Hnxystep][Hnxyt]
          double *qgdnv,    // [Hnvar][Hnxystep][Hnxyt]
          long *sgnm)       // [Hnxystep][narray]
{

  #pragma omp target device (cuda) copy_deps
  #pragma omp task input ([Hnvar][Hnxystep][Hnxyt]qleft, \
                          [Hnvar][Hnxystep][Hnxyt]qright, \
                          [Hnxystep][narray]sgnm) \
                  output([Hnvar][Hnxystep][Hnxyt]qgdnv)
  {

    dim3 block, grid;

    SetBlockDims(Hnxyt * slices, 192, block, grid);

    Loop1KcuRiemann <<< grid, block >>> ((narray, Hsmallr, Hsmallc, Hgamma,
            Hniter_riemann, Hnvar, Hnxyt, slices, Hnxystep,
            qleft, qright, qgdnv, sgnm ));

    cudaDeviceSynchronize();

    if (Hnvar > IP + 1) {
      Loop10KcuRiemann <<< grid, block >>> (narray, Hnvar, Hnxyt,
                  slices, Hnxystep, qleft, qright, qgdnv, sgnm);
      cudaDeviceSynchronize();
    }
  }
}
```

We thus decided to clean up both the *cuRiemann* routine and the kernels it uses by passing as arguments to the kernel what has been received as arguments by *cuRiemann*. The result is a reduction or 29 lines of the original code that get substituted by 8 lines in the OmpSs CUDA version within the *cuRiemann.cu* file. Of course we eliminated the explicit data transfer inside this routine, but also the other data transfers that are explicitly done in the original CUDA program at an outer level. A proper comparison of the total number of lines should be delayed till a version leveraging all the CUDA kernels is done.

We have followed a specific incremental path to move from the pure MPI version to a version that uses multiple MPI processes each of them potentially using multiple threads and GPUs. Looking back to the general structure of the code, it is quite similar to the original pure MPI code. We have used three levels of nested tasks. Alternative task structures could be derived by eliminating some of the pragmas and reducing the nesting level. Also a larger

part of the main loop could be moved to the GPU leveraging other available kernels. As long as the outputs of one task are used as inputs by the next, the runtime should keep data in the GPU avoiding the per tasks data transfer of all its inputs and outputs.

3.3 Results

In this section we present some results and analyses of the behavior of the different versions. A first analysis is done on the OmpSs version with three tasks used as preliminary step for the CUDA version. The results of this version are also representative of those for the single task version. We then look at the CUDA version.

3.3.1 Task splitting

We first show in Figure 6 the timeline of the tasks for a run with 8 MPI processes and 3 OmpSs threads per process and different block sized for 256 on top to 32 on the bottom.

We see the three internal tasks in red, pink and brown respectively. Pink corresponds to the *riemann* call. In the timeline on top we see 4 instances of *j* loop separated by the MPI communication in the make boundary call. We can also see that the actual order of the matrix traversal is twice in one direction and twice in another although in the syntactic structure of the actual code one such instance is encapsulated in a function called *hydro_godunov* irrespective of the traversal direction.

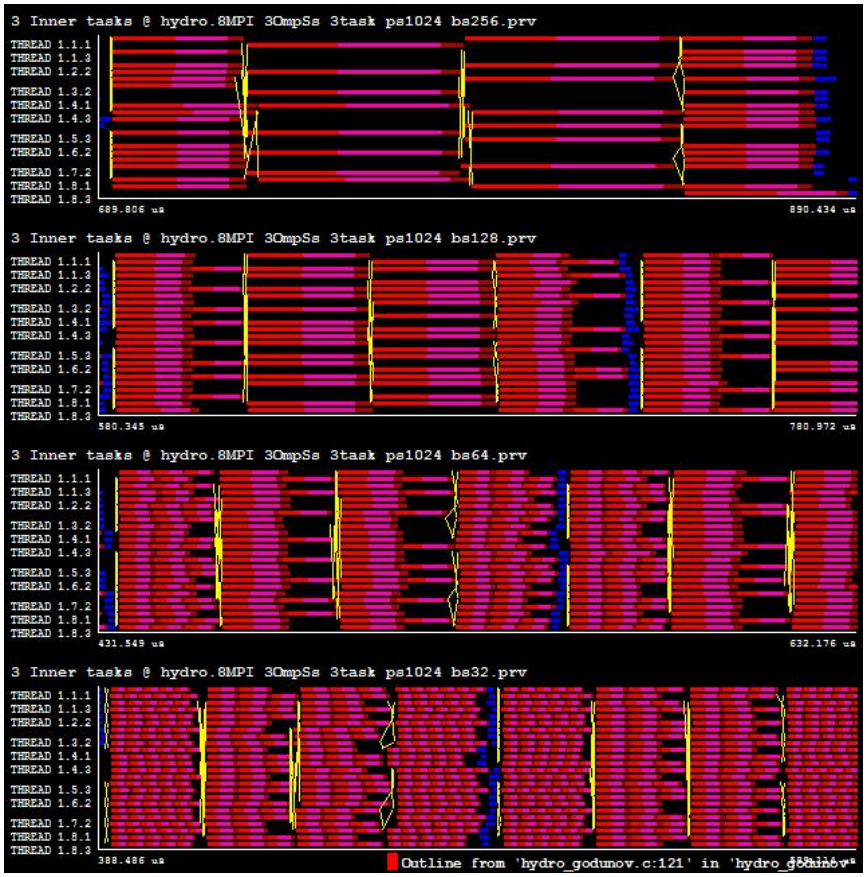

Figure 6: Three internal tasks (red first, pink riemann, brown last) for block sizes between 256 on top and 32 on bottom

We see that for one of the directions and large block sizes, there are not enough blocks to feed the 3 threads in the process while for the other dimension the load balance is better. This has to do with the actual problem size and how data is split into the MPI processes. In this case rectangular blocks are allocated to each MPI process as its number is not a perfect square. When the number of shared memory blocks is not multiple of the number of cores, imbalance appears. The larger the block size, the more imbalanced will be the shared memory execution. We could have certainly computed in the program the block size depending on the size of MPI data structure for the dimension of interest and on the number of OmpSs cores per process. We are essentially interested in understanding the issues so we did not further follow that path in this study.

3.3.2 MPI + OmpSs + CUDA

Figure 7 shows a timeline of one and a half iteration of the run with two processes each of them with one CPU core and 2 GPUs. The GPU tasks are extremely fast compared to the CPU task. We can also see how both GPUs are used randomly within each process. This points to a possible optimization in the runtime that only uses the amount of resources of each type that match the demand of the application. In this case, one GPU would be enough.

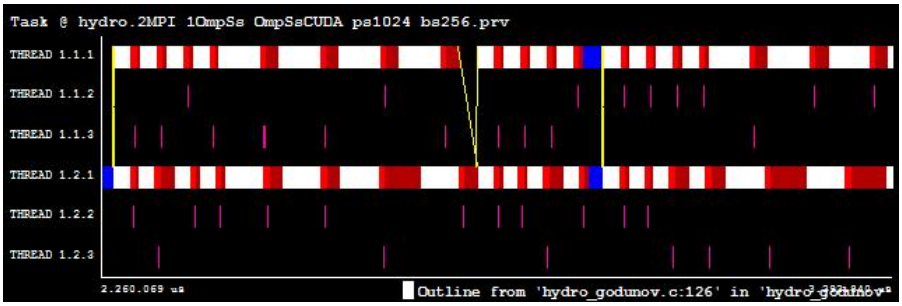

Figure 7: Tasks of MPI+OmpSs+CUDA for a run with one core and 2 GPUs

With a detailed look at the trace (zoom in Figure 8) we can also see that the data transfers between CPU and GPU do represent an important overhead. The bright red in the CPU cores corresponds to the SMP task from within the GPU task is spawned. It can be seen that the time it takes is significantly larger than the actual execution on the GPU, the difference corresponding to the data transfer between CPU and GPU. The use of GPU kernels for other routines and avoiding data transfers would thus be an important line of further code optimization.

Even so, a significant gain has been obtained compared to the single core per process whose timeline at the same scale of Figure 7 is shown in Figure 9.

A final timeline in Figure 10 shows how using several CPU cores (3 in this case) for each process overlaps the execution of the parts that have not been moved to the GPU while a single GPU is shared by the CUDA tasks instantiated by all threads.

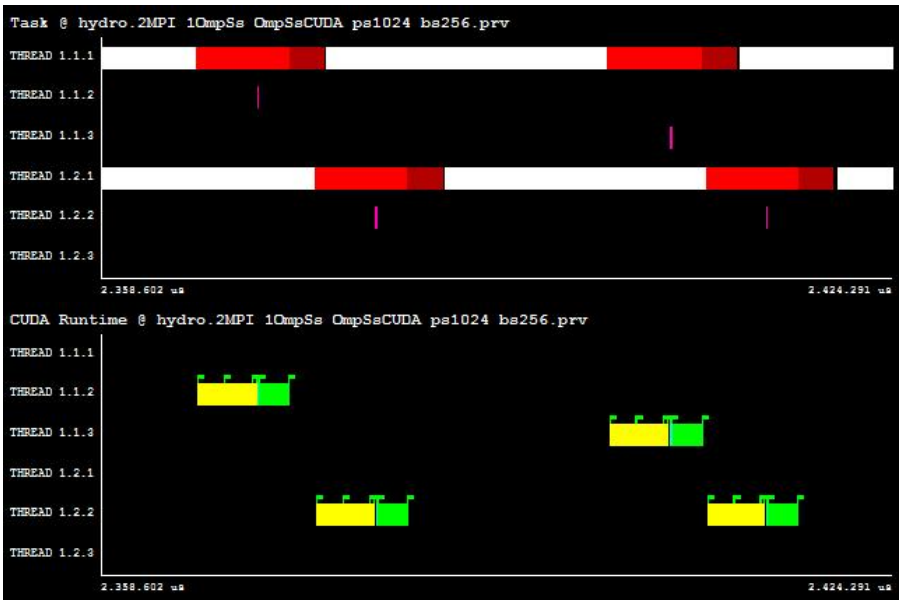

Figure 8: Zoom of Figure 7showing a pair of GPU tasks instantiated by each process (top) and CUDA runtime calls by the Nanos++ runtime to transfer data from CPU to GPU (yellow) and in the reverse direction (green)

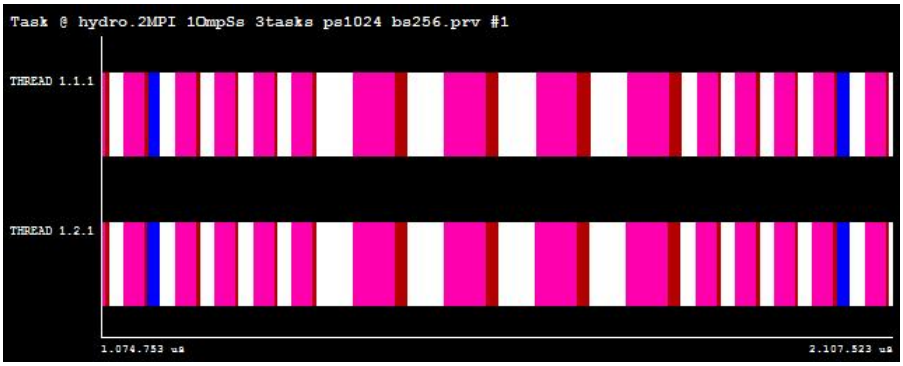

Figure 9: one iteration of the 3 tasks version with just one thread for each of the two MPI

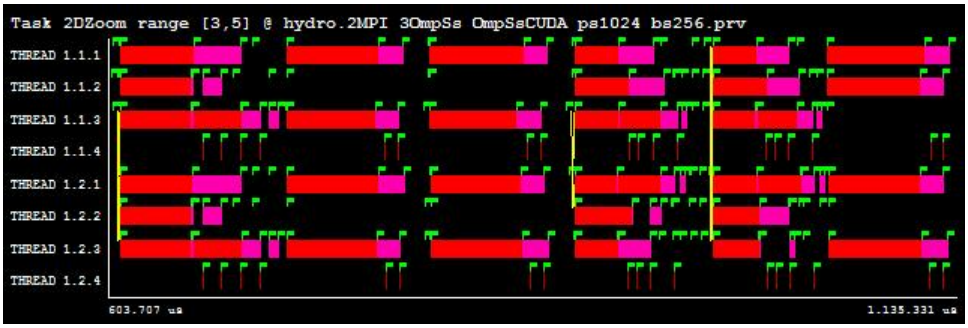

Figure 10: execution with 3 SMP threads sharing one GPU

3.4 Feedback on MPI/OmpSs

Two types of observations or recommendations can be made from this analysis. A first set of findings are methodological:

- **Top down design**: we would recommend following a top down approach structuring and taskifying an application. A typical objection to a task based or functional parallelization approach is that not enough parallelism may be obtained at the outermost levels. We do believe that parallelizing at all levels through nesting, where each level contributes with potential parallel work will be an important approach to generate work for the huge amount of cores that future systems will have.

- **Importance of blocked code structure**: blocking is useful for locality purposes, but also to generate enough granularity in the tasks to amortize the overheads of the runtime. In our case, the code restructuring of the latest version provided by CEA aiming at manually introducing CUDA was actually doing blocking, that ended up being very useful.

- **Importance of proper declaration of variables**: it is important to properly declare arrays passed as arguments to functions instead of passing just pointers and doing index arithmetic. This makes the code cleaner and is also necessary to properly specify the directionality clauses in OmpSs pragmas.

- **Importance of memory management**: interaction between code syntactical structure and memory allocation and parallelization is very important. Delaying the binding of objects to an address space and specially avoiding binding to absolute addresses in devices is important to let the runtime decide the mapping of those objects to the physical resources.

A second set of suggestions are more related to the model itself and the implementation:

- **For loops and dependences**: the specification that a for loop is parallel and leave the splitting into tasks to the runtime is a natural approach in OpenMP but is not supported in the task-based dependence model of OmpSs. Of course this is trivial for parallel loops as in OpenMP. The issue with OpmSs is that it specifies inouts of tasks, which means a task must be identified to specify its inouts. In the case of a dynamically scheduled for loop the tasks are not defined till the very moment of executing them. Combining dependences and late specification of tasks is thus an issue which we do believe important and are considering some possibilities, but no final approach has been decided.
- **Refactoring tools** to do loop splitting, and interchange would be very useful. Extracting loops outside of procedures would be an interesting feature.

4 GROMACS

4.1 Short application description

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles.

It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions, but since GROMACS is extremely fast at calculating the non-bonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, e.g. polymers.

4.2 Parallelization approach

We started with an MPI version of the application and added a second level of parallelism with SMPSs. The main target was to improve the load balance of the application. For this reason we focused the parallelization in the more unbalanced areas of the code.

We implemented five SMPSs tasks from the parts of the code identified previously. The main issue we faced was that the different tasks need to perform a global reduction introducing a sharing problem. We evaluated different approaches, like having replicas of the data structure that needed to be accessed; finally we used some locks to provide mutual exclusion to the SMPSs threads executing the tasks.

4.3 Results

In the chart below we are showing the performance results we obtained when running GROMACS in MareNostrum with different number of MPI processes and configurations.

The labels BASE/CYCLIC refer to the distribution of MPI processes between the nodes, the series labeled as BASE are distributed in a lineal way, while the series labeled CYCLIC are distributed round robin. The results show that the impact of distribution of MPI processes depend on the number of MPI processes. Up to 32 MPI processes the performance is better with the BASE distribution, while when running with 64 or more MPI processes the CYCLIC distribution performs better.

The series labeled with LeWI are executed with the Load Balancing Library and the LeWI algorithm compared to the original execution of the MPI+SMPSs application labeled as ORIG. We can see how the time execution is improved always when using the LeWI algorithm.

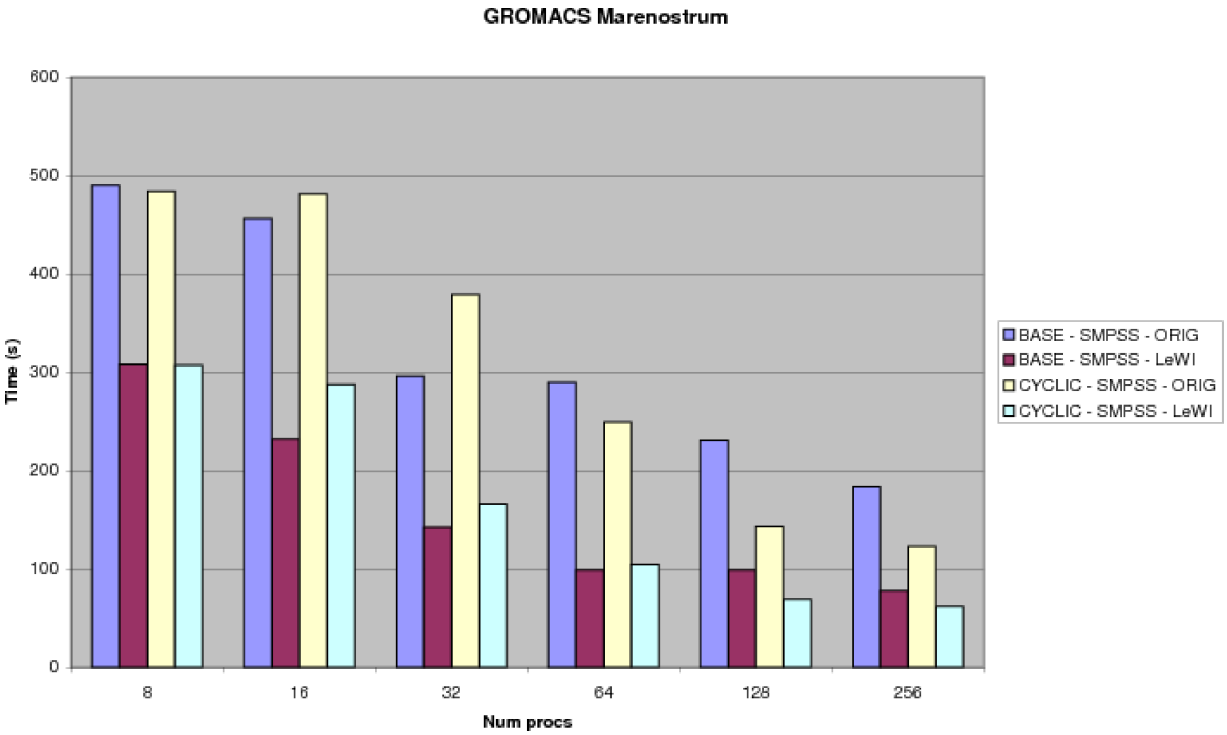

Figure 11: Performance of the GROMACS MPI and MPI+SMPSs versions with and without the LeWI load balancing mechanism.

In both applications the target was to improve the performance of the application without parallelizing the whole code, we have shown how we reduced the execution time for both applications while modifying a small amount of code.

Figure 12: Concurrent execution of coarse grain (non parallelized functions/loops) and fine grain tasks

An interesting experience during the development of this port was that it was not necessary to fully parallelize al the functions that were invoked in sequence in order to compute the force vector. As shown in Figure 12 it was possible to leave sequential one of the three regions of code (the red task in Figure 12) as the use of the SMPSs reduction clause would allow it to

overlap with the other two contributing regions that certainly are parallelized with finer grain tasks (green and blue tasks). By giving high priority to the coarse task, the runtime is able to schedule it as soon as possible and fill the spare time and rest of the cores with the finer grain tasks.

4.4 Feedback on MPI/SMPSs

Some observations that we think are worth highlighting are:

- **Reductions**: This code showed the high importance of reduction operations on large vectors though indirections. Including in the model mechanisms to express flexible reduction patterns and semantics (like leveraging the User Defined Reductions from OpenMP) and providing implementations that support then efficiently when applied with indirections on large vectors would significantly help programmability and reduce development cost.

- **Fighting Amdahl's Law**: This work raised a very optimistic message that it is still possible to leave regions of the original code unparallelized as long as it is possible to overlap them with regions that are parallelized with fine grain. This is not the case with pure fork join codes where it is necessary to fully parallelize each individual code section.

5 GADGET

5.1 Short application description

Gadget is a production code that performs a cosmological N-body/SPH simulation. It can be used to address different astrophysical problems such as colliding and merging galaxies or the formation of large-scale structure in the Universe.

The gravitational forces in Gadget are computed with a hierarchical tree algorithm and the fluids are represented by means of smoothed particle hydrodynamics (SPH). The computation is performed by time-steps.

5.2 Parallelization approach

The version we started with was parallelized with MPI. Although the application comes with its own load balancing code that dynamically updates the tree, there were still some imbalance problems that were not solved.

The parallelization with SMPSs was aimed at solving this imbalance while using the Load Balancing Library DLB with the LeWI algorithm. For this reason it was not necessary to parallelize the whole application as we could obtain a performance improvement with a partial parallelization of the code and the load balancing algorithm (LeWI) when running in the same amount of computational resources.

We identified and parallelized 3 loops in a code with more than 35.000 lines of code. These loops were the ones with more potential because of its load and its imbalance. But we are aware that there are more loops in the application that follow the same pattern and can be parallelized in a similar way. The loops parallelized with SMPSs where the files density.c, gractree.c, and hydra.c.

We used blocking in all the loops to provide the SMPSs tasks enough granularity and at the same time have enough malleability to perform the load balancing. The main change in the code was the following transformation to allow blocking as shown for one of them in Figure 13. On top we have the original code and on bottom the blocked code. In all the cases we moved the necessary code inside the taskified functions. And the global variables used by the tasks changed to private or local ones.

```
for(nexport = 0, ndone = 0;
    i < N_gas && nexport <All.BunchSizeHydro-NTask;
    i++) {

                                    (a)

for(nexport = 0;
    ii<N_gas && nexport < (All.BunchSizeHydro - (BS * NTask));
    ii+=BS){
    int tope=MIN(N_gas, ii+BS);
    hydro_force_block_task(ii, tope...);
                                    (b)
```

Figure 13: blocking transformation in GADGET. (a) original loop, (b) blocked loop

5.3 Results

In the following Figure we show some of the results we obtained with one of the inputs of the application. We executed GADGET with different configurations. The version of the code is indicated by ORIG/SMPSS, ORIG means the original application with the MPI parallelization and SMPSs refers to the MPI+SMPSs version of the code parallelized by us.

The series labeled with LeWI are executed with the dynamic load balancing library. The labels BASE/CYCLIC refer to the distribution of the MPI processes across the nodes, as we saw that there was a meaningful difference in the performance obtained depending on how were distributes the MPI processes. The MPI processes in the series labeled BASE were distributed sequentially, while the MPI processes in the series CYCLIC were distributed in round robin.

The experiments were executed in Marenostrum with 256 MPI processes in nodes with 4 cores with shared memory. All the executions were done with the same computational resources, in the Figure we are showing the execution time for each time-step of the run.

We can see how the distribution of MPI processes between nodes is the most important factor in the execution time. And the SMPSs version with the Load balancing algorithm LeWI obtains the best performance results when using any of the two distributions of MPI processes.

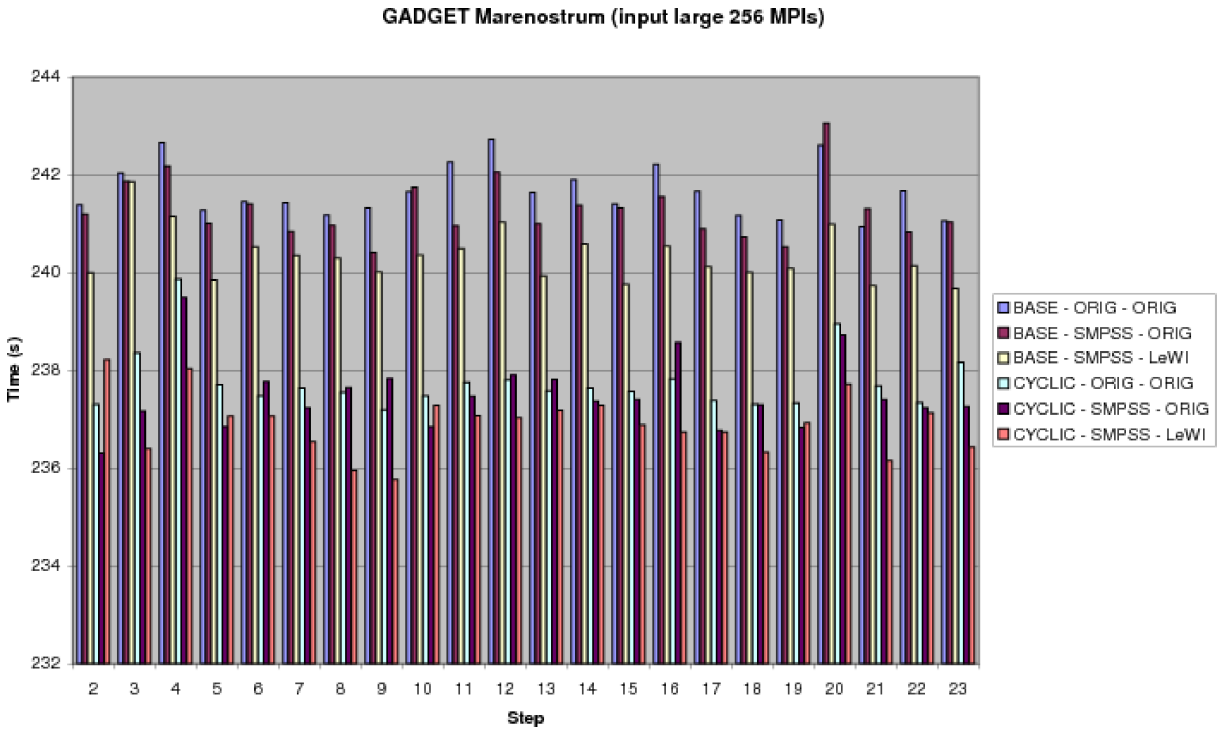

Figure 14: Performance of the GADGET MPI and MPI+SMPSs versions with and without the LeWI load balancing mechanism.

5.4 Feedback on MPI/SMPSs

Some observations that we think are worth highlighting are:

- **Focus on the load imbalanced regions**: For applications that expose load imbalance, a partial SMPSs parallelization focusing just on the regions that expose it is a pragmatic first step that leads to good performance benefits with limited effort even for very large codes. Attempting a full parallelization of the whole program and overlap of computation and communication can be far more expensive and if not the main bottleneck, such an asynchronous execution will not really pay off.

- **Interleave mapping of processes to nodes**: This does have a significant impact on the potential gain as our DLB mechanism can only balance the load between processes within a node. If the load is not well balanced at the node level the DLB mechanism cannot get any gain. Interleaved mappings of processes to nodes is thus recommended as it is frequent that contiguous ranks have similar loads.

6 CP2K

6.1 Short application description

CP2K [1] is a freely available program (under the GNU Public Licence), written in Fortran 95, to perform atomistic and molecular simulations of solid state, liquid, molecular and biological systems. It provides a general framework for different methods such as density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW) known as QUICKSTEP [2], and classical pair and many-body potentials. Recently, linear-scaling DFT and Møller-Plesset 2nd order perturbation (MP2) methods have been added, broadening the applicability of the code to a wider range of users. On top of these energy evaluation methods, a large variety of simulation tools are built, including Molecular Dynamics, Monte Carlo, Geometry Optimisation, Path Integrals, Nudged Elastic Band and Normal Mode Analysis.

CP2K is a popular and important code in Europe, used by research groups in Materials Science, Computational Chemistry and Biomolecular Sciences. It is the third most heavily used code on the UK national HPC service 'HECToR', dominates usage of the Cray XE systems at CSCS, Switzerland, and is available to PRACE users on the JUGENE, CURIE and HERMIT Tier-0 systems.

6.2 Parallelization approach

Amounting to over 600,000 lines of code, CP2K is designed as a flexible and extensible framework for implementing a variety of atomistic simulation methods. As such, the subroutines that dominate the runtime of a particular execution of the code vary greatly depending on what type of simulation is being performed. As a result there are several important distributed data structures, including 3D grids (which may be distributed in one, two or three dimensions depending on how many processors are employed and the overall grid size), sparse and dense matrices, and distributed task lists. A detailed discussion of the parallelization strategy and performance aspects of the code can be found in prior work by Bethune [3].

In order to ensure good scalability on modern multi-core HPC platforms, a mixed-mode MPI/OpenMP approach has been taken, and as a result scalability on over 10,000 CPU cores has been obtained [4]. One aspect of this work has been the introduction of a bespoke sparse matrix library into CP2K. Called DBCSR (Distributed Block Compressed Sparse Row), this library takes advantage of the block structure of matrices used in CP2K arising from the use of a localized Gaussian basis set, to provide efficient and scalable matrix multiplication, addition, and other primitive operations. DBCSR has been used to implement a new linear scaling DFT algorithm (LS_DFT)[5], based solely on density matrix operations, and so simulations based on this method typically spend over 90% of the time in the DBCSR matrix multiplication kernel. Table 1 below shows the profile of an LS_DFT calculation on BlueGene/P showing, taken from [6]. The dominance of DBCSR routines is clearly seen.

	1024		2048		4096		8192	
Total job time	11009.24 s		5512.72 s		3033.99 s		1809.77 s	
CP2K run time	10803.34 s		5495.54 s		2790.11 s		1462.35 s	
cp_dbcsr_mult_NS_NS	10512.06 s	97.30%	5347.552 s	97.31%	2701.518 s	96.82%	1414.34 s	96.72%
cp_dbcsr_frobenius_norm	90.04 s	0.83%	47.46 s	0.86%	28.987 s	1.04%	14.238 s	0.97%
cp_dbcsr_get_occupation	85.03 s	0.79%	39.225 s	0.71%	25.535 s	0.92%	12.114 s	0.83%
calculate_rho_elec	52.99 s	0.49%	26.524 s	0.48%	13.271 s	0.48%	6.647 s	0.45%
integrate_v_rspace	22.71 s	0.21%	11.373 s	0.21%	5.707 s	0.20%	2.846 s	0.19%

Table 1: Profile by function for a linear scaling DFT calculation of 6144 atoms using MOLOPT basis set

In DBCSR, matrices are distributed over MPI processes in a 2D grid. If the matrix is sparse, but not uniformly so, load balance is achieved by a permutation of the rows and columns of the matrix, so that each process receives the same number of non-zero blocks. For simplicity, the following description will assume a square matrix and square processor grid, although in practice, non-square systems can assign multiple domains per processor in one dimension so that there are in total the same number of domains in each dimension. Multiplication is carried out following Cannon's algorithm – if there are $P = P_x^2$ processes in each dimension then there are P_x local multiplications carried out, and P_x-1 row and column-wise shifts, with the results of the local multiplications accumulated into the result matrix, as shown in

Figure 15.

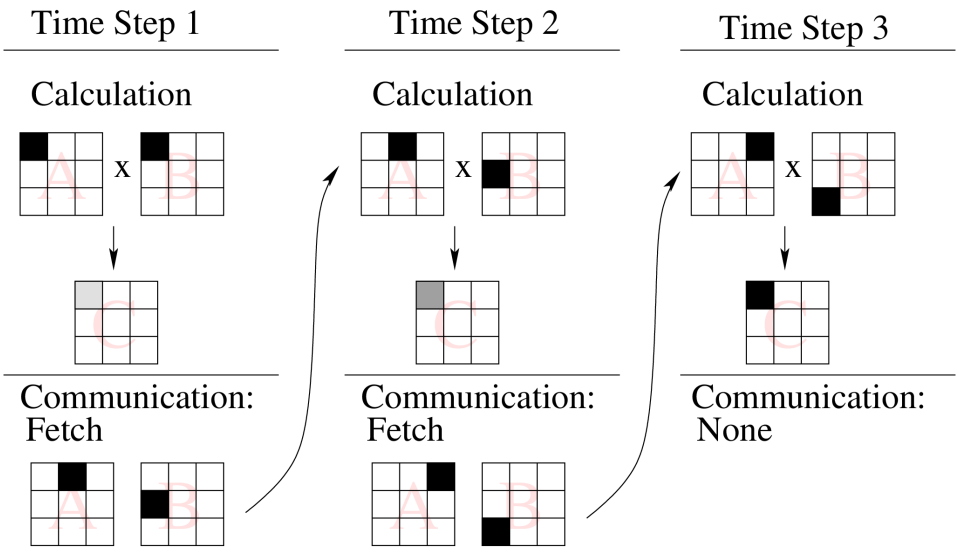

Figure 15: Schematic of Cannon's algorithm in CP2K (Image courtesy Urban Borstnik, University of Zurich)

The local multiplications are carried out by a cache-oblivious recursive strategy. For a local multiplication of matrix blocks `C = C + A x B`, where `C` is an N by M matrix, `A` is N by K and `B` is K by M, then to perform the multiplication we recursively subdivide the N, M and K indices, until the remaining number of non-zero elements is below a certain terminating value at which point the recursion terminates and the sub-block is multiplied directly.

In common with the rest of the code, there is an also OpenMP parallelization layer in DBCSR. The current implementation is very straightforward – we simply decompose the

matrix by row and assign a contiguous block of rows to each thread. Each thread then builds a separate recursion tree, and forms the product of its rows of `A` with columns of `B` and accumulates these into `C`. While this solution entirely avoids the need for any synchronization between threads (as each writes to a disjoint subset of the rows of `C`), it does suffer from one serious drawback – load imbalance. While it is perfectly possible to divide the rows up over threads such that each has the same number of total non-zero elements to multiply, recall that in Cannon's algorithm we repeatedly shift `A` and `B` while accumulating partial products into `C`. As a result, we must maintain the same decomposition of rows to threads to avoid having to either perform an expensive merging of the thread-local copies of C after each step (to allow us to change the decomposition of rows to threads per step of Cannon's algorithm). Thus, even if the decomposition is chosen to balance the work per thread across the entire global row of the matrix, we cannot guarantee load balance at each individual step of Cannon's algorithm.

To overcome this problem we proposed abandoning the row decomposition and instead form a single recursion tree at each stage of the Cannon's multiplication, dividing the multiplications at the leaves of the tree between threads using a task-based approach. While this can clearly be done using OpenMP tasks (with appropriate locking to protect concurrent access to regions of `C`), we felt this could also be efficiently implemented using SMPSs, since correctly respecting data dependencies would obviate the need for locking inside tasks. In addition, communication is currently done outside the OpenMP parallel region, but with SMPSs there is scope for including the row and column shifts directly as tasks (with appropriate dependencies) which might allow for more efficient overlapping of communication and local multiplication. We have not had time to implement this, and all our results do not include communication but compare only the local multiplication. Below we describe several different implementations of the DBCSR multipication using both SMPSs and OpenMP tasks, and compare the performance and programmability of both approaches.

6.3 Results

Rather than implement the new parallelization directly within CP2K, which we anticipated would be difficult due to the Fortran 90 usage restrictions – particularly with respect to modules, assumed shape arrays and internal subprograms – we instead used a test program which was originally written during the development of DBCSR. This essentially carried out only the local recursive multiplication step, but does not include the MPI shifting between processes, and as such is an ideal platform for comparing different shared memory parallelization schemes.

The following versions were implemented, and the performance of each was tested for up to 32 CPU cores (`OMP_NUM_THREADS = 32`, `CSS_NUM_CPUS = 32`) on a Cray XE6, with two AMD Opteron 'Interlagos' 2.3 GHz 16-core CPUs forming a single shared memory node. In all cases we have used a 2000 by 2000 matrix, which is a typical local block size for problems of interest in CP2K. Initially we have used a dense matrix where all elements are non-zero for testing purposes, as it allows us to compare the efficiency of the different implementations directly.

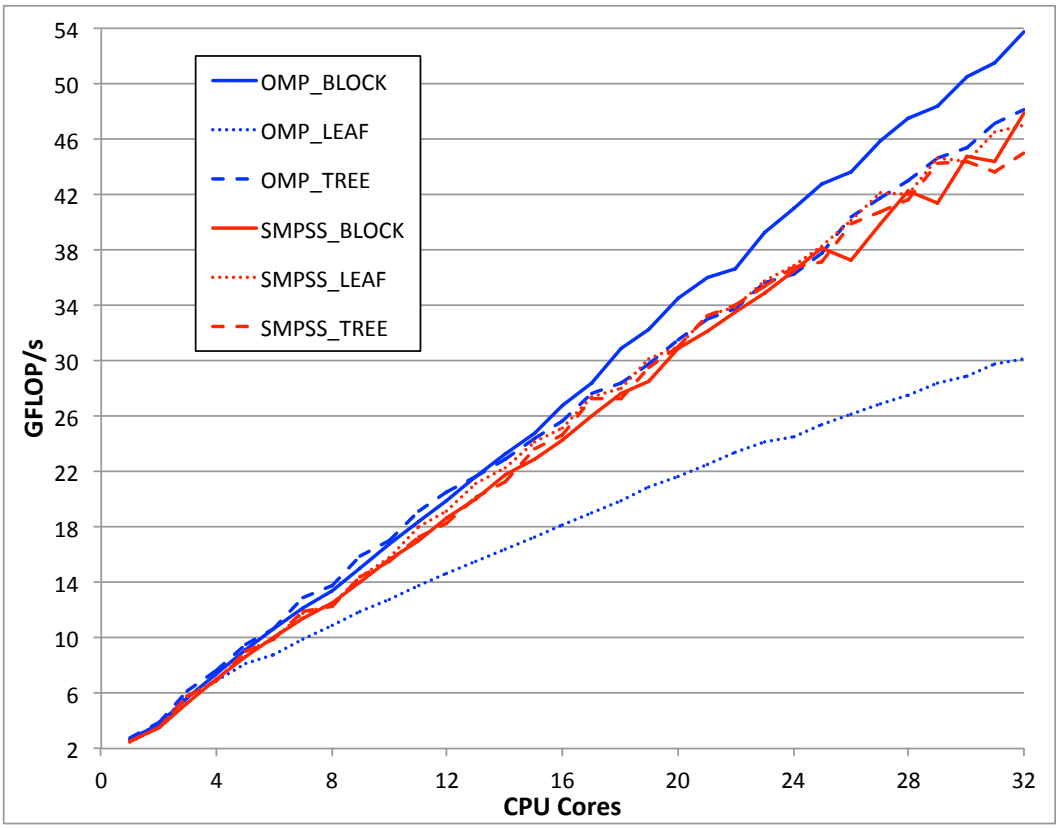

Figure 16: Performance against number of cores for dense matrix multiplication using OpenMP and SMPSs

OMP_BLOCK: This closely matches the current implementation in CP2K, where the matrix is partitioned into contiguous blocks of rows for each thread. Clearly in this case where there are many more rows than threads, load imbalance is not a significant issue even when the number of threads does not exactly divide the number of rows, and this scales close to linearly.

OMP_LEAF: Here we create an OpenMP task at each leaf of the recursion tree. We create a 2-dimensional array of locks sized so that each task takes a single lock, protecting access to the region of `C` to which it writes. However, as shown in Figure 16, the performance of this method is very poor due to the overhead of taking and releasing the locks inside each task. The performance still scales close to linearly, indicating the lock collisions are relatively infrequent as there are many more locks than threads.

OMP_TREE: To rectify the cost of locking in the OMP_LEAF method, instead of creating OpenMP tasks (and doing the locking) at the leafs, we instead create tasks at a certain depth of the tree, and take the locks corresponding to the entire region of `C` which will be updated by the sub-tree generated by that task. The choice of where to generate the tasks is a compromise between having enough tasks that lock collisions become suitably unlikely, without introducing too much task creation and execution overhead. After some experimentation, we found that creating approximately 10 tasks per thread gives the best performance, that is tasks were generated at depth `d` where `d ~= log2(nthreads*10)`.

SMPSS_BLOCK: This is exactly the same as the OMP_BLOCK implementation, except that each block is encapsulated within an SMPSs task. We see that again this gives close to

linear scaling as expected, although the absolute performance is slightly lower than that achieved with OpenMP.

SMPSS_LEAF: For both the remaining SMPSs implementations we required a method of ensuring that the data dependencies on `C` are respected. Conceptually this is an `inout` dependency on the region of `C` written by the task, but there are two problems with this in practice. Firstly the entire data array of the matrix is passed as an argument to the task, since the sparse indexing relies on being able to access elements by an offset from the beginning of the array. Secondly, as we cut up the M and N indices each task processes a 2D sub-block of the 2D array, which is not a pattern for which SMPSs can correctly handle dependencies. To overcome these limitations, we create a second 2-dimensional array, which does not contain any data but is used to enforce the dependencies, and the real data array is passed as an `input` parameter to the task. The new array is sized such that a task generated at the shortest path through the tree is passed a 1x1 sub-block of this array, representing the larger 2D block of the data array that it will write to. Tasks generated further down the tree are passed the same 1x1 sub-block, even though they may in fact write to a smaller 2D data block. This requirement for being over-cautious in the dependencies is a result of the fact that tasks may be generated at different tree levels (in the dense case the level differs only by 1, but it may be more in the general sparse case). Having set up this second array, in the SMPSS_LEAF implementation we simply create an SMPSs task at each leaf of the tree. Compared to the OMP_LEAF implementation this gives much better performance, since there is no locking overhead inside each task, and even with 32 CPUs there are enough independent tasks to keep all CPUs busy.

SMPSS_TREE: Similarly to OpenMP, we also implemented generation of SMPSs tasks at a fixed depth during the descent of the tree. However, despite experimenting with the chosen depth, we found that the performance could not be improved over the SMPSS_LEAF implementation, indicating that the overhead of task creation, execution and completion is not a significant effect in this case.

These results show that for simple implementations, using SMPSs tasks with data dependencies is significantly more efficient that using locking with OpenMP tasks. However, by careful implementation in OpenMP to reduce the impact for locking, comparable performance with SMPSs can be achieved.

For dense, well load-balanced matrices, the existing block-decomposition approach is sufficient. However, to show the advantage of the task-based parallelism over the static decomposition we created load imbalanced system by removing the rows 1 to N/4 from the matrix. Clearly this is a very artificial system, but it does allow us to clearly compare the performance under non-ideal conditions. We tested the two BLOCK implementations and also the best task-based ones – OMP_TREE and SMPSS_LEAF, and the performance is shown in Figure 17. Here we can clearly see that the BLOCK implementations give worse performance (by a factor of ¼ compared to the fully dense case) while the tasked implementations (both SMPSs and OpenMP) are almost entirely unaffected. There is a small performance impact in these cases due to the fact that some indexing calculations are still performed for the empty blocks, which takes a small amount of time. Nevertheless, we can clearly see here that using dynamic load balancing with tasks provides a significant advantage over the statically load balanced row decomposition.

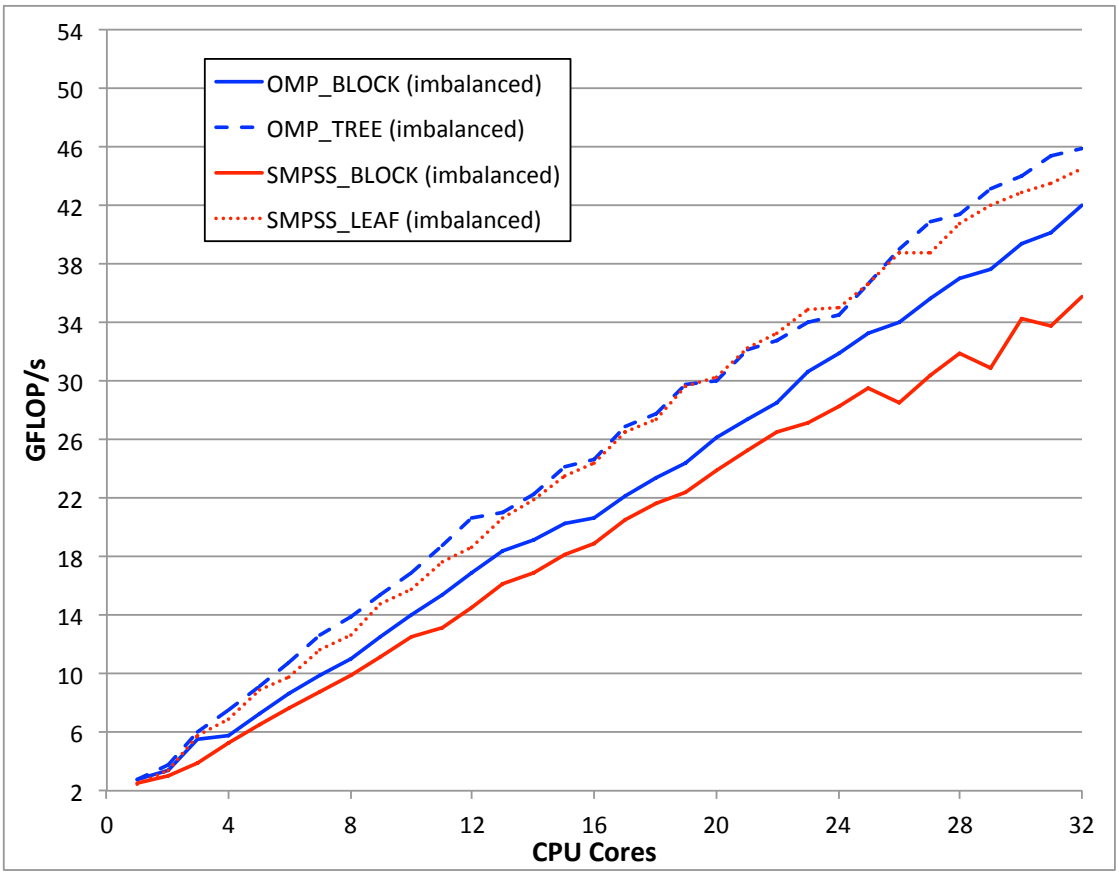

Figure 17: Performance of selected implementations with a load-imbalanced sparse matrix

.

6.4 Feedback on MPI/SMPSs

While we found the performance of SMPSs to be as efficient as OpenMP, there are several issues that would prevent us from adopting SMPSs within CP2K itself.

- **Restrictions on the features of Fortran 90 that can be used with SMPSs** – CP2K makes extensive use of modules, Fortran POINTER types, and internal subroutines. Even within our test program we had to remove the subroutines we wanted to turn into tasks from modules, resulting in difficult to maintain code. This approach would not be practical in the full code.

- **Requiring Fortran INTERFACE blocks for each SMPSs task** – this essentially requires that the parameter list of the task be duplicated in the calling subroutine. This is poor software engineering practise, as it means that any further changes to the taskified subroutine's argument list must now be made it two or more places.

- **Dependencies on multi-dimensional subarrays** – while it may be difficult to implement in the runtime, the inability to do this resulted in significant complications to our code.
- **Lack of conditional tasks** – in OpenMP it is possible to create a task if a certain condition is met. In this case this was used to create tasks at a particular depth of the

tree. In SMPSs to achieve the same effect, we needed to create a new wrapper subroutine that was taskified, and calls the original subroutine internally. This again requires duplication of code, and reduces code clarity.

- **Subroutine-scope of tasks** – related to the above, OpenMP tasks can be added with less disruption to the layout and flow of existing code since arbitrary structured blocks of code can be designated as tasks, whereas in SMPSs manual 'outlining' of the task body as a subroutine is required.

Many of these issues seem to be Fortran-specific and the C SMPSs interface seems significantly more mature and well used. Since a significant fraction of current HPC codes are written in Fortran, we recommend addressing some of the above issues in future releases of SMPSs if it is to become more attractive to a wider range of HPC developers.

7 MRGENESIS

7.1 Short application description

MRGENESIS is an extension of the RMHD (Relativistic MagnetoHydroDynamic) version of the RHD (Relativistic HydroDynamic) code GENESIS. It has been developed by the Department of Astronomy and Astrophysics, at University of Valencia. It is written in Fortran 90.

The code performs simulations of RMHD flows, like collisions of magnetized shells and the radiation resulting from these collisions by using the 1D model of MRGENESIS.

7.2 Parallelization approach

Figure 18shows a flowchart of the main loop of the application. Each loop has three MPI communication blocks (halo), before each heavy compute block (step). Then, a value dt is computed (tstep3) and reduced (dt reduce) between each MPI process.

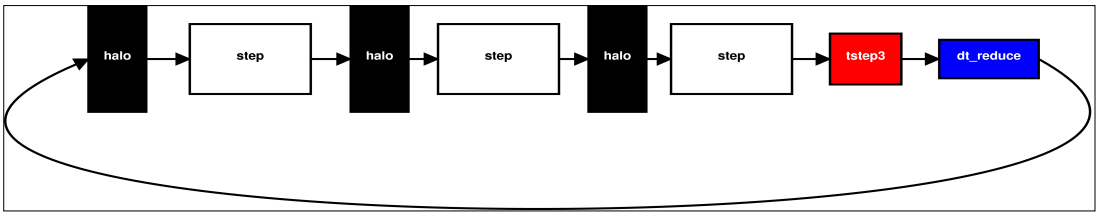

Figure 18: Algorithm flow

In the original version of the algorithm, only blocks step and tstep3 are parallelized using the OpenMP model. With the use of SMPSs, dt_reduce can be also parallelized as a background task while other serial code is executed before restarting the main loop. Blocks of communication halo will remain serialized.

Figure 19 shows a viable distribution of the grid among MPI processes, and how they communicate with each other.

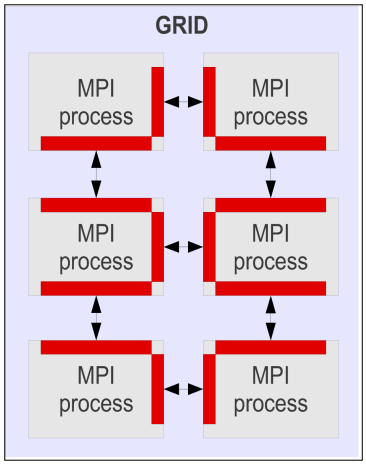

Figure 19: MPI distribution

7.3 Results

The defined tasks will allow the parallelization of the blocks with a high level of computation

Figure 20 shows a Paraver trace. The trace shows an iteration of the main loop, where all tasks has been defined; also it is shown a zoom to see in detail the last task.

One representative thing of the trace is the small amount of MPI communication. It was shown before in Figure 19, where only the boundary part of every grid distribution was transferred among processes. Now, the trace shows the three blocks of halo_communicate, with only a few individual communications among the processes. It could have been possible to parallelize these communications and fit them into SMPSs tasks so the computation could overlap the communication, but the time invested in these individual communications is not large enough to justify the extra overhead that it would introduce.

Figure 21 gathers the speedup of the application for each programming model after every optimization has been applied. As it happened with the original code, the MPI version stops scaling at 512 processors, while the two hybrid parallel models keep scaling at the maximum value of resources.

The differences of performance between the two hybrid programming models were not very clear in the previous chart because of the resolution. Figure 22 shows the real improvement of SMPSs versus OpenMP for each configuration of processes.

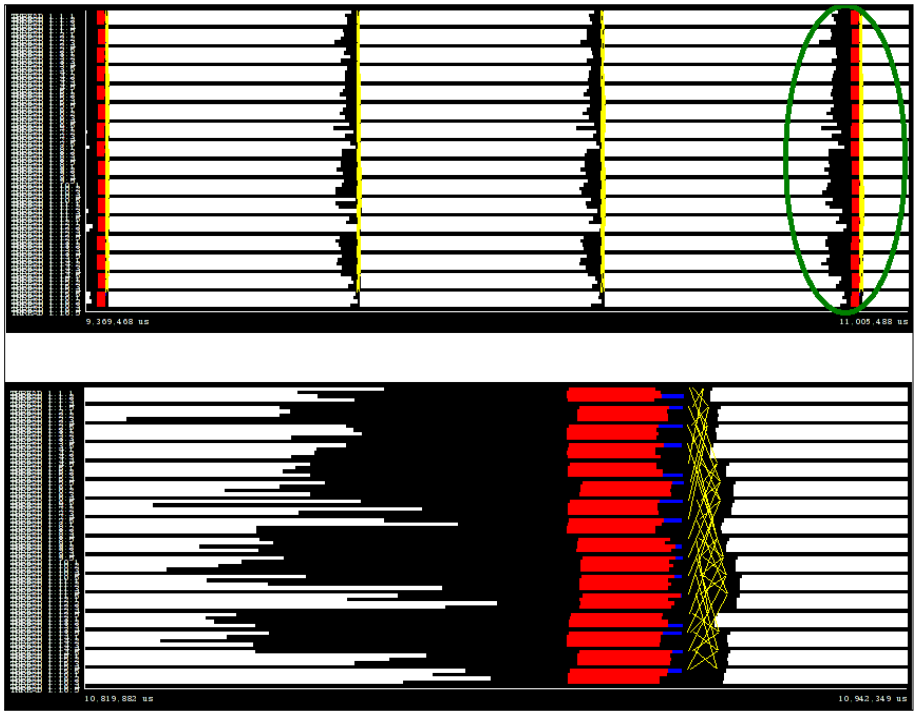

Figure 20: Trace of a MRGENESIS execution using this resources: 16 MPI processes and 4 SMPSs dedicated threads for each process. A total of 64 cores.

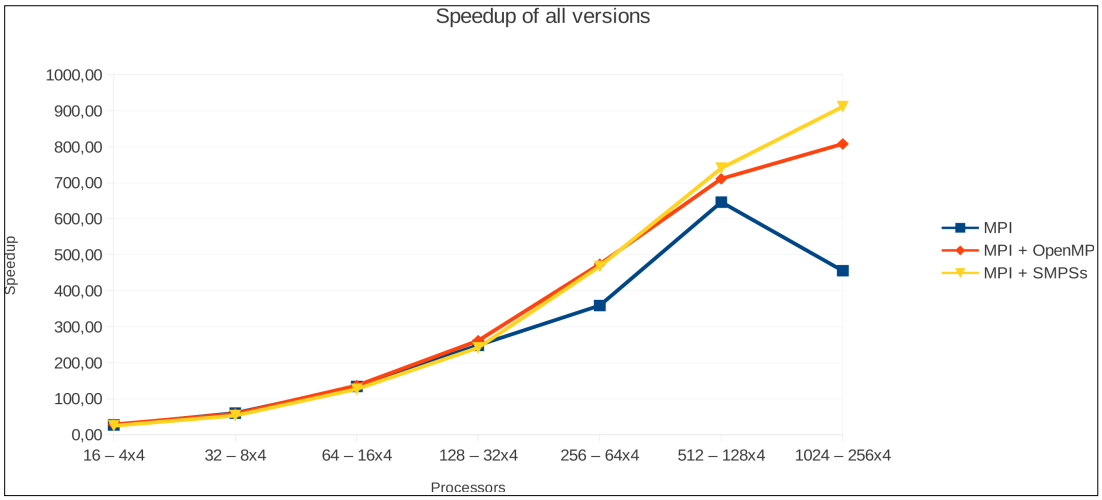

Figure 21: Speedup of each programming model

In this case, it is appreciated that OpenMP gets better performance than SMPSs using few resources. As we increase the number of processes, the SMPSs performance surpasses the other model.

Figure 22: Improvement of SMPSs versus OpenMP

7.4 Feedback on MPI/SMPSs

Both hybrid models, OpenMP and SMPSs, achieve a very good performance over MPI, mainly because the chosen tasks are blocks of code with some huge computation.

The major issue faced has been how to efficiently manage the memory associated to every task. In the original OpenMP code, all the memory structures were global structures accessed by everyone. But, in order to get the best potential of SMPSs, all these structures has been privatized, so the runtime can know the dependencies between them.

As a result of this, SMPSs is getting better scalability than OpenMP; though, with few resources OpenMP gets better performance.

8 Shallow Water

8.1 Short application description

The NCAR shallow water model [9] solves the shallow water equations with a second-order finite-difference code [10]. The equations are solved over a regular rectangular domain with periodic boundary conditions in order to emulate a sphere. Details of the numerical scheme can be found in [11].
The shallow water equations are similar to the equations solved by many atmosphere models, and so shallow can be a useful simplification for certain parts of a full climate model, in particular data-parallel grid based computations and halo-swap communications.

The original NCAR code base provided only a 1D MPI decomposition, so the code was rewritten in a modular fashion with modern Fortran 90 features and a 2D domain decomposition strategy.

8.2 Parallelization approach

The motivation for exploring SMPSS on this simple shallow water HPC benchmark code comes primarily from our involvement in a U.K. joint NERC/Met Office-funded project, called Gung Ho (http://www.met.reading.ac.uk/Data/CurrentWeather/wcd/blog/gungho-development-of-a-new-dynamical-core-for-the-unified-model/). The aim of this project is to develop a new highly scalable atmospheric dynamical core for the Met Office's Unified Model of Weather and Climate prediction for the emerging many-core high petascale (and eventually exascale) computers.

Task-based parallelism is one approach that has shown promise in increasing the scalability of scientific codes, in particular in the Linear Algebra domain. Hence the interest in seeing how this approach might work on the shallow code. Shallow has relatively simple data access patterns and a straightforward domain decomposition strategy, that leads to an efficient MPI solution. The problem is 2D so communication is not a big issue and when the data per core fits into the level 2 cache, the performance is very high. Thus, it was not expected that an SMPSs implementation would outperform the MPI implementation, rather we were interested in how easy it would be to implement the code in SMPSs and in the performance one could achieve using the task-based approach in SMPSs.

The shallow water calculations are performed on the following fields:

- Velocities u and v
- Pressure P
- Mass fluxes U and V
- Potential vorticity Z
- Field height H

The basic algorithm is shown in Figure 23

The SMPSs version was created from an existing C version of the code. It was restructured so that each field is blocked internally, and each task performs calculations for one block of one or more fields. The boundary exchange was taskified accordingly.

```
init psi, p
init u, v
init uold, vold, pold
for ncycle {
     compute cu, cv, z, h
     update boundaries cu, cv, z, h
     compute unew, vnew, pnew
     update boundaries unew, vnew, pnew
     time smoothing uold, vold, pold
     update for next cycle u/unew, v/vnew,
p/pnew
}
```

Figure 23: basic structure of the Shallow waters code

8.3 Results

The tests were run on a four processor (quad core AMD Opteron 8378, 2.4 GHz) shared memory system with 24GB RAM. For the MPI version the processes were were (scatter) pinned to cores to improve data affinity. Figure 24 shows the results for the MPI version of the shallow water code and Figure 25for the SMPSs version.

The SMPSs versions were improved by introducing artificial task dependencies in order to avoid renaming.

The performance difference between the MPI version and the SMPSs version is primarily attributed to cache affinity.

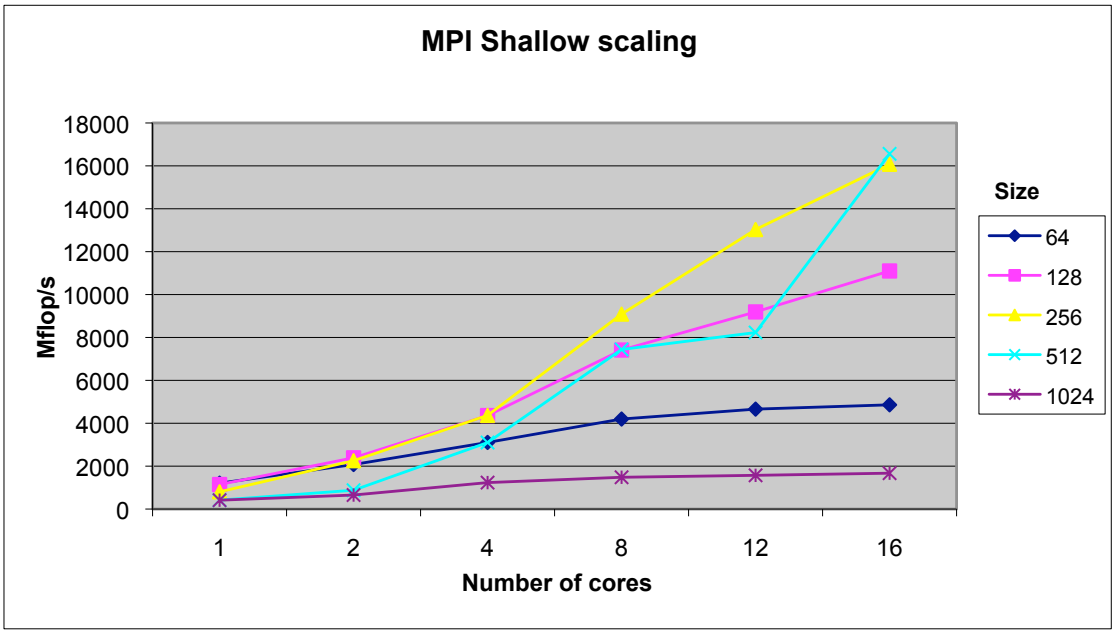

Figure 24: MPI Shallow scaling on 1 to 16 cores

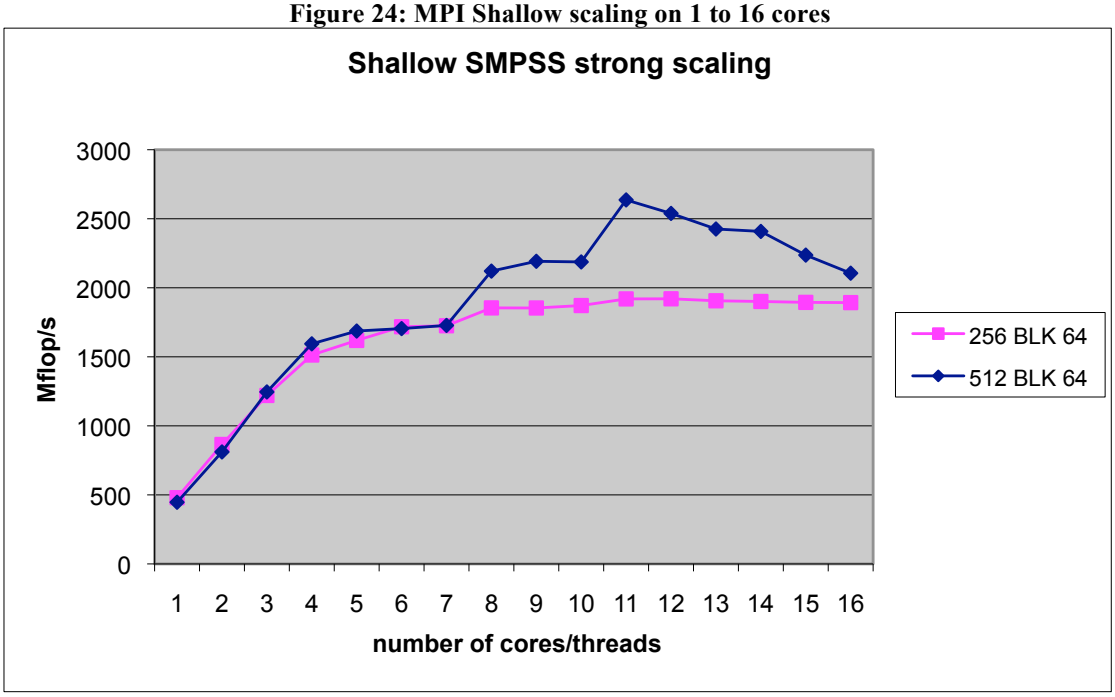

Figure 25: Shallow SMPSs scaling on 1 to 16 cores (64 * 64 block size per task)

8.4 Feedback on MPI/SMPSs

It was relatively simple to produce the SMPSs versions with explicit blocking of the original shallow water code.

The biggest hinderence to high performance in the best SMPSs implementation developed is due to the loss of cache affinity between tasks. Improved affinity scheduling should help to reach the same level of performance as the MPI version.

9 Conclusions and future work

The applicability of the StarSs model (SMPSs and OmpSs) versions has been investigated to different depth levels for several applications. This work has been very useful from the very practical issues of detecting bugs in the implementations and help to correct them to a more fundamental issue of identifying programming methodology aspects and detecting the appropriateness of the model itself. Some of the problems in the SMPSs model and implementation are really important. They have caused more cost in the porting effort than what should be reasonable and we are grateful for the interaction/cooperation with original code owners and developers involved in this activity. We think OmpSs is already addressing them (or will in a very short term). In any case, this feedback will be very useful for guiding the continuous evolution of the OmpSs model and implementation.

Some of the examples have shown performance improvements over the MPI case, but in many others the results are similar to those obtained with OpenMP. Beyond the assumed higher overheads, StarSs does have conceptual benefits over pure OpenMP both in terms of concurrency and asynchrony on one side and data management and locality on the other. Some applications may inherently have little potential gain for some or both of these concepts. In some other cases, the StarSs parallelization may follow very closely the fork join parallelization of OpenMP and thus no gain is obtained. The programmability and flexibility gains in some cases (i.e. GPUs) are important although often the actual appreciation is very subjective. As of today, in some of these codes that already use OpenMP the SMPSs performance achieved by the relatively limited effort in this task does not justify a change in the application. We believe that further work could demonstrate in many of the cases that the claimed joint programmability and performance gains are possible.

We do believe that this type of activity is extremely important. For the above reasons and the usefulness of the results we think that more should be done. We also consider that it will be necessary to create repositories of example codes making then publicly available as far as possible as examples and training material that could be used to expand proper practices.

10 References

[1] CP2K Project website, http://www.cp2k.org

[2] Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassaing and J. Hutter, Comp. Phys. Comm. 167, 103 (2005).

[3] Improving the performance of CP2K on the Cray XT, I. Bethune, 2010, Proceedings of the Cray User Group (CUG 2010)

[4] CP2K: High Performance Computing, http://www.nanosim.mat.ethz.ch/research/CP2K

[5] Linear scaling self-consistent field calculations for millions of atoms in the condensed phase, J. VandeVondele, U. Borstnik and J. Hutter, J. Chem. Theory Comput. (2012)

[6] Million Atom KS-DFT with CP2K, I. Bethune, A. Carter, X. Guo and P.Korosoglou, PRACE White Papers (2012), http://www.prace-ri.eu/IMG/pdf/cp2k.pdf

[7] HYDRO, Pierre-François Lavalléea, Guillaume Colin de Verdièreb, Philippe Wauteleta, Dimitri, Lecasa, Jean-Michel Dupaysa. PRACE White Paper. Available online at www.prace-ri.eu

[8] RAMSES. http://web.me.com/romain.teyssier/Site/RAMSES.html - Last accessed May, 2012.

[9] UCAR. NCAR HPC shallow water model tutorial, October 2006. http://www.cisl.ucar.edu/docs/hpc_modeling/

[10] R. Sadourny. The dynamics of finite-difference models of the shallow-water equations. Journal of the Atmospheric Sciences, 32:680–689, 1975.

[11] G-R. Hoffmann, P.N. Swartztrauber and R.A. Sweet. Aspects of Using Multiprocessors for Meteorological Modelling. In, Multiprocessing in Meteorological Models, Eds. G-R Hoffmann and D.F. Snelling. Springer-Verlag Berlin Heidelberg, 1988.